Application Note

Interfacing the TMS320C6X DSP to the PCI bus using the V360EPC Controller

1.0 Objective

This application note describes how to interface Texas Instrument's TMS320C62x/C67x Digital Signal Processor (DSP) to the PCI bus using the V360EPC (Enhanced PCI Controller) from V3 Semiconductor.

Throughout this document, references are made to the operation of the V360EPC and TMS320C6201. Basic familiarity with these devices is assumed. If you do not have the relevant data sheets or User's Manual for the V360EPC, you may download them from the V3 Semiconductor web site at:

http://www.vcubed.com/products/chips/epc.htm

2.0 Overview

The TMS320C62x/C67x has two independent interfaces: the Host Processor Interface (HPI) and the External Memory Interface (EMIF). On the EMIF bus, we have used asynchronous SRAM as shared memory between the DSP and PCI. Other memories, such as SDRAM and SBSRAM can also be accessed by the DSP via the EMIF bus.

A PCI bus agent is able to access both the shared memory (SRAM) and the DSP's internal memory (CPU internal memory). Accessing the SRAM is accomplished by using the EMIF external bus. An access to the internal memory is accomplished by accessing the HPI.

The DSP is able to access the shared memory through its EMIF interface. It is also able to access a PCI slave agent through the same EMIF interface. The V360EPC bridge from V3 semiconductor provides the highest performance

V3 SEMICONDUCTOR
THE EMBEDDED INTELLIGENCE COMPANY

V3 Semiconductor Corp.
2348G Walsh Avenue
Santa Clara, CA 95051
Phone (408) 988-1050, Fax (408) 988-2601
Toll Free (800) 488-8410 (US and Canada)

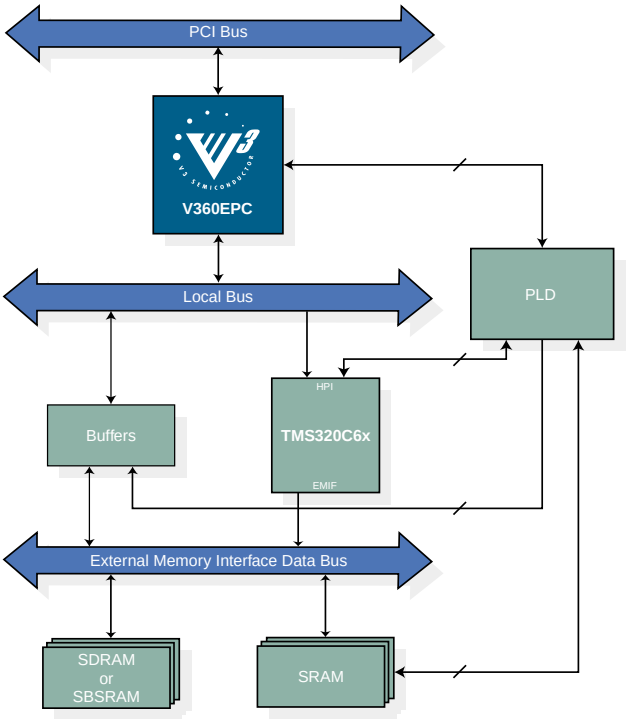
http://www.vcubed.com

PCI bridge for this application. Two possible designs are shown in the figures that follow: Figure 1 illustrates a high-performance application where the PCI agent can access the HPI interface at the same time the EMIF is accessing its external bus. The buffers between the two buses provide isolation. These buffers are very useful when this type of concurrence is crucial for high performance. Figure 2 illustrates a simple design where HPI and EMIF concurrence is not important to the application.

As shown in Figure 1, the EMIF accesses the PCI bus through the EPC. Accesses from the EMIF to EPC can easily be accomplished by enabling the buffers to the EMIF–to–EPC direction. Accesses from the V360EPC to the EMIF external bus (to shared memory) may also be accomplished by enabling the buffers to the other direction (EPC–to–EMIF). This application uses three SN54ABT16601 18-bit buffers from Texas Instruments. See Interconnection.

The buffers are enabled for either EPC–SRAM access or EMIF–EPC access. For EPC–SRAM access, the EMIF relinquishes its external bus (HOLD, HOLDA protocol). In the case of EMIF–EPC access, the EMIF waits until the EPC relinquishes the local bus, if the EPC is currently accessing the HPI bus; otherwise, buffers may be opened immediately.

Figure 1: High Performance Design Using Buffers



When the buffers are closed, the following is true:

- The buffers are tristated on both buses.
- The EPC–HPI access is enabled when the EPC requests it and the EMIF is not requesting it.
- The EMIF-to-external bus is enabled by default.

Because concurrence is not needed, the design shown in Figure 2 omits this option. When the PCI agent requires access to the HPI, the EPC sends a request to the EMIF to relinquish its external bus. In this design there is no need for buffers to isolate the two buses as only one master at a time is driving the bus.

In an application where PCI to HPI accesses are limited, the design shown in Figure 2 is easier and more cost-effective to implement. Please note the following:

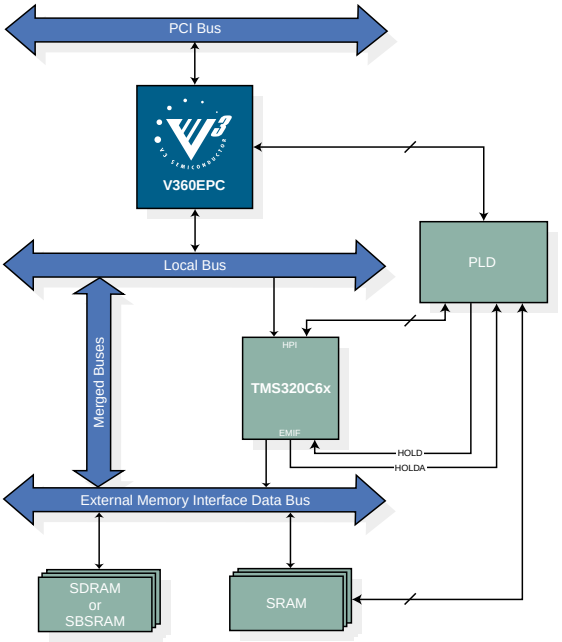
- PCI to HPI also holds the EMIF bus (as does HOLDA).
- EMIF access to any of its local bus memory peripherals (SRAM, SDRAM, or SBSRAM) holds the EPC from accessing HPI or SRAM. The EPC does not get $\overline{\text{LBGNT}}$ until the EMIF is done.

• Waveforms for the direct connect design variation are not included.

2.1 V360EPC Overview

EPC stands for *Enhanced PCI Controller.* The V360EPC can act as PCI bus master, PCI slave, or PCI host bridge. In this application, the DSP acts as the PCI master, controlling the PCI bus through the EMIF bus.The EPC acts as the EMIF target on the EPC's local bus.

Figure 2: Direct Connect Design



The EPC is a PCI target when an agent on PCI bus is accessing the HPI interface or the shared memory. For the shared memory accesses, the EPC requests the EMIF external bus; the EMIF relinquishes the bus when it is ready. The DSP is parked on the EMIF bus unless an external master (like the EPC) requests ownership of the bus.

The EPC has two programmable apertures for accesses from PCI to the local bus. One aperture is programmed to access the HPI while the other is programmed to access the shared memory. The apertures are size programmable for up 2 Gbytes. They are also bus size programmable to 8-, 16-, or 32-bit bus size. For accessing the HPI (which is a 16-bit wide bus) an aperture can comfortably be programmed to 16-bit bus in the **LB_SIZE** register. Accesses to the SRAM can be programmed to 32-bit bus wide, which is the default setting in the registers.

An access to the aperture can be as long as a 1 KB burst. The V360EPC has 640 bytes of programmable FIFOs to accommodate bursts. An access from PCI to local (or local to PCI) may also be established with the same aperture performance levels through either of the EPC's two DMA engines. Setting the DMA channels allows large blocks of data to be transferred to the shared memory or to the CPU's internal memory space.

The EPC has two programmable apertures for accesses from local to PCI. PCI space can be mapped to $\overline{CE0}$ in the EMIF memory space. An EMIF access to the $\overline{CE0}$ region is decoded as local-to-PCI access by the EPC. You may program other regions in the EMIF space to be decoded as local-to-PCI access by using the other local-to-PCI aperture. For example, $\overline{CE0}$ local to PCI aperure0, CE2 local to PCI apertue1.

The V360EPC has demultiplexed address and data buses which follow the AMD29k interface protocol. The EPC can run at up to 50 MHz (up 40 MHz at 3.3 volts). It is bootable through the EEPROM, the PCI bus, or the local processor. The EPC is fully compliant with the PCI 2.1 Target Specification and is Hot Swap Capable according to the PICMG™ Hot Swap Specification.

2.2 TMS320C6x Overview

The TMS320C6x has two independent interfaces: the HPI—Host Processor Interface and the EMIF—External Memory Interface.

HPI—Host Processor Interface

The HPI is a target-only asynchronous interface through which the host processor accesses the CPU's internal memory space. In this application, the host processor could be a PCI agent which accesses the HPI through the V360EPC PCI bridge.The HPI is a 16-bit wide data bus, HD[15:0].

> **Note:** Due to the 32-bit word structure of the DSP architecture, all transfers with the host consist of two consecutive 16-bit halfwords.

The HPI accesses to the internal CPU space are through three registers: the HPIA (Host Processor Interface Address) register, the HPID (Host Processor Interface Data) register, and the HPIC (Host Processor Interface Controller) register.An access to any of the registers must consist of two consecutive 16-bit accesses. The HPI does not have an address bus, instead it has an address register HPIA. The address register HPIA provides the address for any transfer from the HPI to the internal memory space. The HPIA can be programmed to increment for each access to the HPID. The data (from the HPID register) and the address (from the HPIA register) is transferred to the internal CPU memory space through the DMA auxiliary channel. Both the data and address registers can only be accessed through the HPI interface. The HPIC controller register can be accessed through either the HPI or the DSP's CPU. Two pins, HCNTL[1:0], determines which HPI register is accessed according to Table 1below.

Table 1: Access to HPI Registers

HCNTL[1:0]	Description
00	Access to HPIC
01	Access to HPIA
10	Access to HPID with HPIA postincrement
11	Access to HPID without HPIA postincrement

To read or write to the internal memory, a typical access should start with setting the HPIC register, followed by the HPIA address register, followed by accesses to the HPID. For further information about the HPI please refer to TMS320C6201/C6701 Peripheral Reference Guide (literature number SPRU190) from Texas Instruments.

Table 2 describes all of the HPI interface signals.

Table 2: HPI Interface Signals

Signal	Type	Description
HD[15:0]	I/O	16-bit data bus.
R/$\overline{W}$	I	Read/not-write signal.

Table 2: HPI Interface Signals

Signal	Type	Description
$\overline{\text{HDS1}}$, $\overline{\text{HDS12}}$	I	Strobe signals. An internal strobe signal is activated when either of these signals is active and $\overline{\text{HCS}}$ is also active.
$\overline{\text{HCS}}$	I	Acts like chip select. Controls strobe signals functions in conjunction with $\overline{\text{HDS1}}$, and $\overline{\text{HDS2}}$ strobe signals.
$\overline{\text{HAS}}$	I	Address latch enable (could be tied high—see the TI datasheet).
$\overline{\text{HBE[1:0]}}$	I	Byte enables.
$\overline{\text{HRDY}}$	O	Host Asynchronous ready output.
HINT	O	Host interrupt output.

EMIF—External Memory Interface

The External Memory Interface is used by the CPU to interface off-chip memory. The EMIF supports a variety of external devices:

- Synchronous burst SRAM (SBSRAM) running at 1 × and 1/2 × CPU clock rate.
- Synchronous DRAM (SDRAM) running at 1/2 × CPU clock rate.
- Asynchronous devices including asynchronous SRAM, ROM, and FIFOs.

The EMIF responds to requests on the external bus from four requesters:

- The program memory controller that services the CPU program fetches on the DSP.
- The data memory controller that services CPU data fetches on the DSP.
- The DSP's DMA controller.
- An external shared memory device.

The last option allows access to the shared memory from the PCI bus. An external device can request the external bus from EMIF by asserting $\overline{\text{HOLD}}$ input. In this application note, the V360EPC requests the external bus to access SRAM. When the EMIF is ready, it asserts $\overline{\text{HOLDA}}$. All EMIF control and data outputs are tristated at this stage.

The EMIF has both generic shared signals and specific control signals to accommodate each external device. Table 3 shows the shared signals for all external devices.

Four programmable memory spaces can be accessed through $\overline{\text{CE}}$[3:0] chip enable. For the current application, program the EMIF CE Space Control Register so that $\overline{\text{CE0}}$ corresponds to SRAM memory and $\overline{\text{CE1}}$ corresponds to the PCI bus (local-to-PCI accesses).

Table 3: EMIF Shared Signals

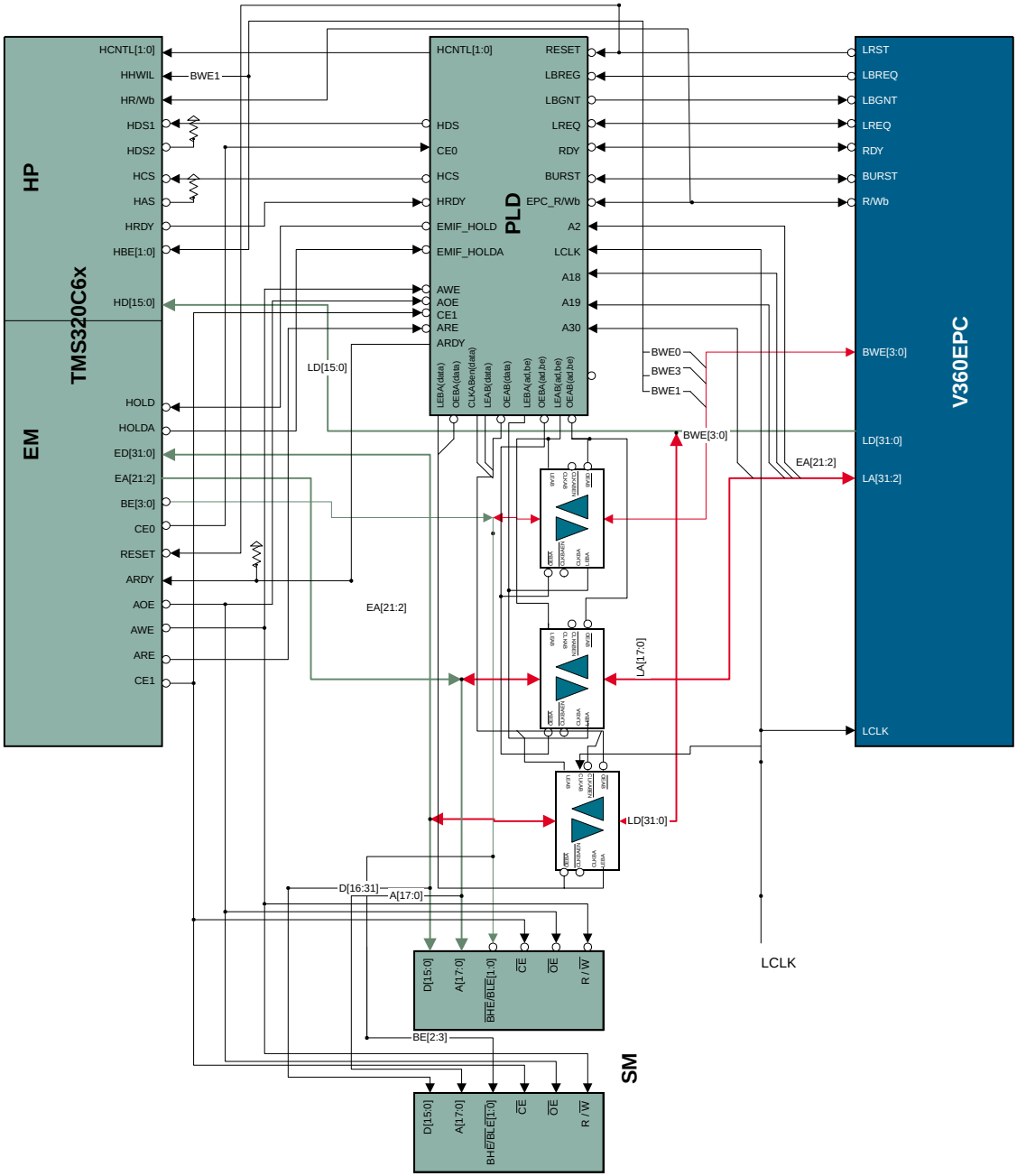
Signal	Type	Description
ED[31:0]	I/O	32-bit data input/output from external memories and peripherals.
EA[21:2]	O	External address output.
$\overline{CE}$[3:0]	O	Active low chip select for memory for the four memory spaces.
$\overline{BE}$[3:0]	O	Byte enables.

The signals listed in Table 4 are specific to the asynchronous external device interface and are in addition to the shared signals shown in Table 3 above.

Table 4: EMIF Asynchronous Interface Signals

Signal	Type	Description
ARDY	I	Ready. Active high asynchronous ready input used to insert wait states for slow memories and peripherals.
$\overline{AOE}$	O	Active low output enable.
$\overline{AWE}$	O	Active low output write strobe.
$\overline{ARE}$	O	Active low read strobe.

Figure 3: TMS320C6201 to V360EPC Interconnection



3.0 Interconnection

The interface between the TMS320C6201 and the V360EPC is shown in Figure 3: TMS320C6201 to V360EPC Interconnection. The full description of the signal connections follows.

3.1 V360EPC Local bus Interface Signals Connection

$\overline{\text{LRST}}$

$\overline{\text{LRST}}$ is the EPC's reset input/output. $\overline{\text{LRST}}$ can be initialized to either input (by pulling the RDIR pin low) or output (by pulling the RDIR pin high). In this application, $\overline{\text{LRST}}$ is initialized to be an output on the local bus. This is typical for PCI master, PCI target, or PCI master and target applications. It is not typical for a PCI host application.

For a host application, $\overline{\text{LRST}}$ is programmed as an input. The host's reset output is connected to the EMIF reset input as well as all other system reset pins (including the EPC's $\overline{\text{LRST}}$ input). $\overline{\text{LRST}}$ output is connected to the PLD reset input and to the EMIF reset input as shown in Figure 3.

$\overline{\text{LBREG}}$

$\overline{\text{LBREG}}$ is the EPC's local bus request output pin. In the course of PCI-to-local access through one of the two apertures (or a DMA from PCI to local), the EPC requests the local bus by asserting the $\overline{\text{LBREG}}$ output signal. A local arbiter grants the EPC the local bus by asserting $\overline{\text{LBGNT}}$ input.

$\overline{\text{LREQ}}$

$\overline{\text{LREQ}}$ is the EPC's local request input/output pin. When the EPC is the local bus master, the EPC drives this pin for the whole access period. When the EPC is the local bus target, the local bus master drives this pin. $\overline{\text{LREQ}}$ is connected to the PLD LREQ I/O. The PLD drives this pin for EMIF-to-PCI accesses after the PLD grants the local bus ($\overline{\text{LBGNT}}$ is high). Otherwise this pin is tristated or driven by the EPC.

$\overline{\text{LBGNT}}$

The EPC's local bus grant input pin. When $\overline{\text{LBGNT}}$ is asserted, the EPC is the local bus master and can drive all local bus data, address, and control signals.

$\overline{\text{RDY}}$

The EPC's ready input/output pin. $\overline{\text{RDY}}$ is directly connected to the PLD $\overline{\text{RDY}}$ I/O pin. For EPC-to-EMIF accesses, the $\overline{\text{RDY}}$ pin is an input. (The EPC is local bus master, the EMIF is local bus target.) For EMIF-to-EPC accesses, the $\overline{\text{RDY}}$ pin is an output. Please see the EPC-to-HPI Read Waveform, EPC-to-HPI Write Waveform, EPC-to-SRAM Read Waveform, and EPC-to-SRAM Write Waveform to see when this pin is driven by the PLD.

For EMIF-to-PCI write cycles, the EPC has large write FIFOs which can queue up to 256 words in a sustained burst. The $\overline{RDY}$ signal is asserted for as long as 1 KB burst transfer. This feature provides higher performance local and PCI bus bandwidth.

For read cycles, the EPC has an 8-word prefetch FIFO dedicated to each aperture (16 words for both apertures). The two apertures can have posted read cycles at the same time. While a read access is in the ready stage (i.e. $\overline{RDY}$ is being asserted) and data is driven from the prefetch buffer of one aperture, the other aperture posted read cycle may be in the read prefetch stage on the PCI bus. When the first read access finishes, the second read access continues without the EPC dropping $\overline{LBREQ}$, thus providing back-to-back access. $\overline{RDY}$ is de-asserted to end the first access then driven again for the second read access. This feature is called multiple posted reads.

$\overline{BURST}$

The EPC's burst input/output pin. This pin is driven high for non-burst accesses and low for a burst access. For burst access, it is driven high just one word prior to the last data in the burst. This pin is directly connected to the PLD.

R/ $\overline{W}$

The EPC's read not write input/output pin. This pin is directly connected to the PLD.

$\overline{BWE}$[3:0]

The EPC's byte enables input/output pins. These pins are connected to the EMIF byte enable pins through the buffers for PCI-to-SRAM or EMIF-to-PCI accesses. $\overline{BWE0}$ and $\overline{BWE3}$ are connected to the HPI's BHE and BLE inputs. $\overline{BWE0}$ is BLE (HBE[0]) and $\overline{BWE3}$ is BHE (HBE[1]).

LD[31:0]

The EPC's input/output data bus. The lower 16 bits are connected directly to the HPI data bus via HD[15:0]. LD[31:0] are connected to the EMIF data bus through latching buffers for EMIF-to-PCI or PCI-to-SRAM accesses.

LA[31:2]

The EPC's address bus input/output pins. A30 is connected to the PLD and is used to decode accesses from PCI-to-HPI. A19, A18 and A2 are also used to decode the type of access to the HPI. See Section 5.0—Programming.

The lower address LA[21:2] are connected through the buffers to EMIF EA[21:2] external bus.

LCLK

The EPC's local clock input pin. LCLK is the same clock connected to the data latching buffers and to the PLD. There is no relationship between the TMS320C6x DSP clocks (CLKOUT1, CLKOUT2) and this clock—they are completely asynchronous.

3.2 HPI Interface Interconnect

$\overline{\text{HCNTL}}$[1:0]

The HPI's control inputs pins. Any access to the HPI must be to one of it's three registers: HPIC, HPIA, or HPID. $\overline{\text{HCNTL}}$ input pins decode which of the three registers is being accessed. These two pins are directly connected to the PLD $\overline{\text{HCNTL}}$[1:0] output pins. For more details, see Section 5.0—Programming.

HHWIL

The HPI's halfword input pin. It indicates whether the first or the second halfword is being transferred. An internal control register bit determines whether the first or the second halfword is placed into the most significant halfword of a word. In a 16-bit region, the EPC's BWE1 behaves like A1 (low for the first 16 bits in a word and high for the second halfword), which means they can be directly connected.

HR/ $\overline{\text{W}}$

The HPI's read/not-write input. Although this pin can be directly connected to the EPC's R/ $\overline{\text{W}}$, it needs to be driven from the PLD for this application.

$\overline{\text{HDS1}}$, $\overline{\text{HDS2}}$

The HPI's data strobe input signals. Either signal asserted, together with the assertion of $\overline{\text{HCS}}$, can generate an internal strobe signal. Both of them are not necessary to generate the strobe signal. This is why one of them is driven from the PLD and the other is disabled by a pull up resistor. For this application, $\overline{\text{HDS1}}$ is driven from PLD and $\overline{\text{HDS2}}$ is pulled high.

> **Note:** $\overline{\text{HSTRB}}$ is the strobe signal not $\overline{\text{HDS1}}$ or $\overline{\text{HDS2}}$. When $\overline{\text{HDS1}}$ is asserted there is a delay before $\overline{\text{HSTRB}}$ is asserted. (Refer to the HPI internal signal description in TI's User's Manual.)

$\overline{\text{HCS}}$

The HPI's chip select input pin. $\overline{\text{HCS}}$ has to be asserted to access the HPI. This pin is directly connected to the PLD.

$\overline{\text{HAS}}$

The HPI's address latch enable input pin. For this application, this input is disabled by a pull up resistor.

$\overline{\text{HRDY}}$

The HPI's ready output pin. This pin is directly connected to the PLD.

$\overline{\text{HBE}}$[1:0]

The HPI's byte enable inputs. For a write access, it determines which byte to write in a halfword. For a read access, the HPI reads 16 bits regardless of the byte enable value. $\overline{\text{HBE}}$[0] is connected to BWE0 and $\overline{\text{HBE}}$[1] is connected to BWE3.

HD[15:0]

The HPI's 16-bit data bus I/O. It is directly connected to LD[15:0].

3.3 EMIF Interface Interconnect

The EMIF has three control signal groups:

- Asynchronous Memory controller
- SBSRAM
- SDRAM

Note: This application note only describes the Asynchronous Controller connections—no other memory devices are being used on the EMIF bus. Connecting additional memory devices (SDRAM or SBSRAM) will not affect the design provided they are mapped to different $\overline{\text{CE}}$ space (not $\overline{\text{CE0}}$ or $\overline{\text{CE1}}$).

$\overline{\text{HOLD}}$

The EMIF's bus request input. An external device can request the EMIF bus by asserting this input. For an EPC-to-SRAM access, the PLD asserts this input. This pin is directly connected to the PLD.

$\overline{\text{HOLDA}}$

The EMIF's bus acknowledge output. In order to enable an external bus master to use the EMIF bus, the EMIF relinquishes its bus, tristates all its outputs, and asserts HOLDA. This pin is directly connected to the PLD.

ED[31:0]

The EMIF's 31-bit data bus I/O. This pin is connected to LD[31:0] through the data buffers.

EA[21:2]

The EMIF's 19-bit address bus output. This pin is connected to LA[21:2] through the address buffers.

$\overline{\text{BE}}$[3:0]

The EMIF's 4-bit byte enables output. These pins are connected to $\overline{\text{BWE}}$[3:0] through byte enable buffers.

$\overline{CE}$[3:0]

The EMIF's chip enable output. PCI is mapped to $\overline{CE0}$ and SRAM is mapped to $\overline{CE1}$. $\overline{CE0}$ is connected to the PLD and $\overline{CE1}$ is connected to the SRAM chip enable. $\overline{CE2}$ and $\overline{CE3}$ are not mapped in this application. They are optional and may be mapped to SDRAM, SBSRAM, or a second PCI aperture.

$\overline{RESET}$

Chip reset input. This pin is directly connected to $\overline{LRST}$ output.

ARDY

The EMIF's asynchronous ready output. This pin is directly connected to the PLD.

$\overline{AOE}$

The EMIF's asynchronous output enable. This pin is directly connected to the SRAM's $\overline{OE}$ pin and to the PLD.

$\overline{AWE}$

The EMIF's asynchronous write enable output. This pin is directly connected to the SRAM's $\overline{W}$/R pin.

$\overline{ARE}$

The EMIF's asynchronous read enable output. This pin is directly connected to the PLD.

4.0 Read and Write Waveforms and Waveform Notes

The buffers used in this application are three SN54ABT16601/SN74ABT16601 18-bit Universal Bus Transceiver with 3-state outputs from Texas Instruments.

Note: In Table 5 below, the AB direction is EPC-to-EMIF and BA is EMIF-to-EPC.

Table 5: Transceiver Function Table(A–Input, B–Output)

$\overline{\text{CLKENAB}}$	$\overline{\text{OEAB}}$	LEAB	CLKAB	A	B
X	H	X	X	X	Z
X	L	H	X	L	L
X	L	H	X	H	H
H	L	L	X	X	B0[a]
H	L	L	X	X	B0[a]
L	L	L	rising edge	L	L
L	L	L	rising edge	H	H
L	L	L	L	X	B0[a]
L	L	L	H	X	B0[b]

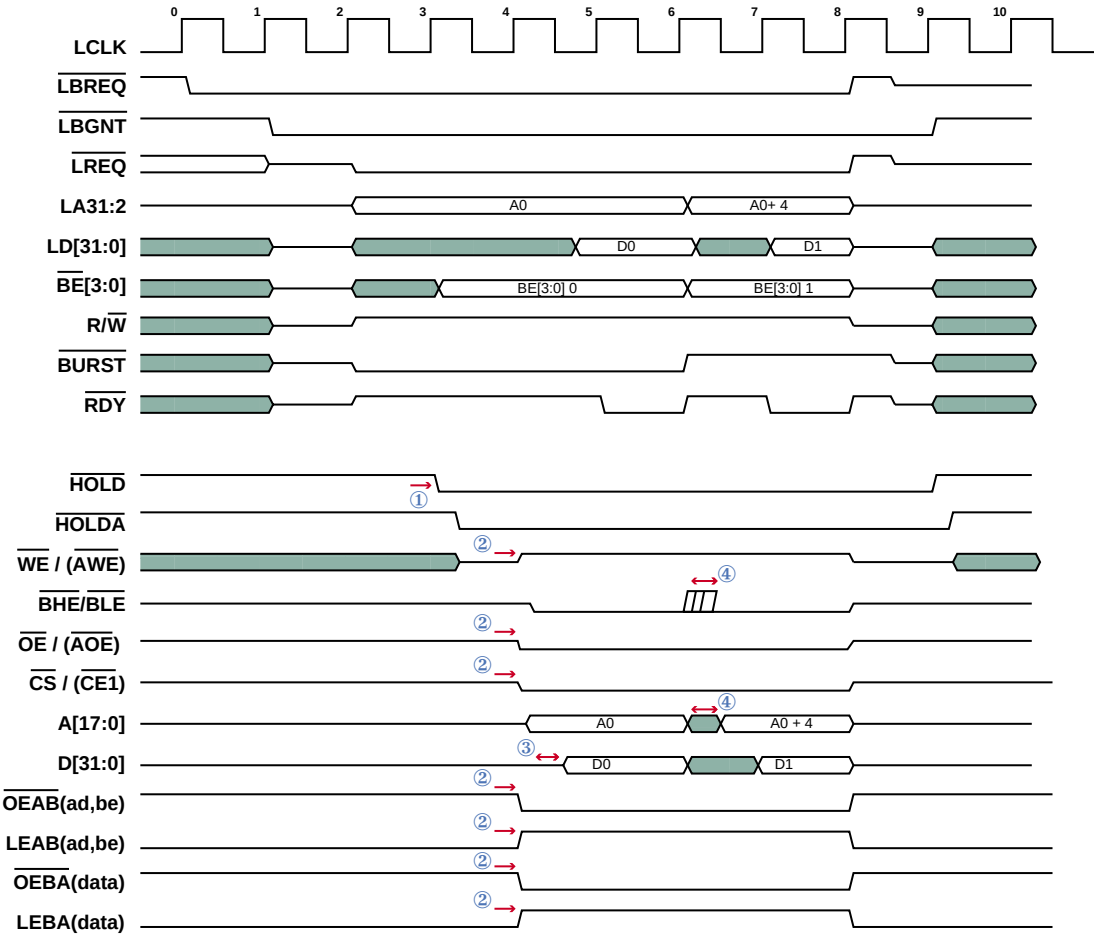
a. Output level before the indicated steady-state input condition were established.

b. Output level before the indicated steady state input conditions were established, provided that CLKAB was low before LEAB went low.

c. The shading indicates functionality used in this application note.

4.1 EPC-to-SRAM Read Waveform

EPC-to-SRAM Read Cycle



EPC-to-SRAM Read Cycle Notes

① At LCLK=2, after the EPC has granted the BUS ($\overline{\text{LBGNT}}$ is asserted low), the EPC starts the cycle by asserting $\overline{\text{LREQ}}$ and driving the address bus. On the rising edge of LCLK=3, the PLD decodes the address to the HPI or SRAM access according to the aperture programming. When it decodes SRAM access, the $\overline{\text{HOLD}}$ signal to EMIF control is asserted low. When the EMIF is ready to give up the external bus it asserts $\overline{\text{HOLDA}}$ asynchronously to LCLK.

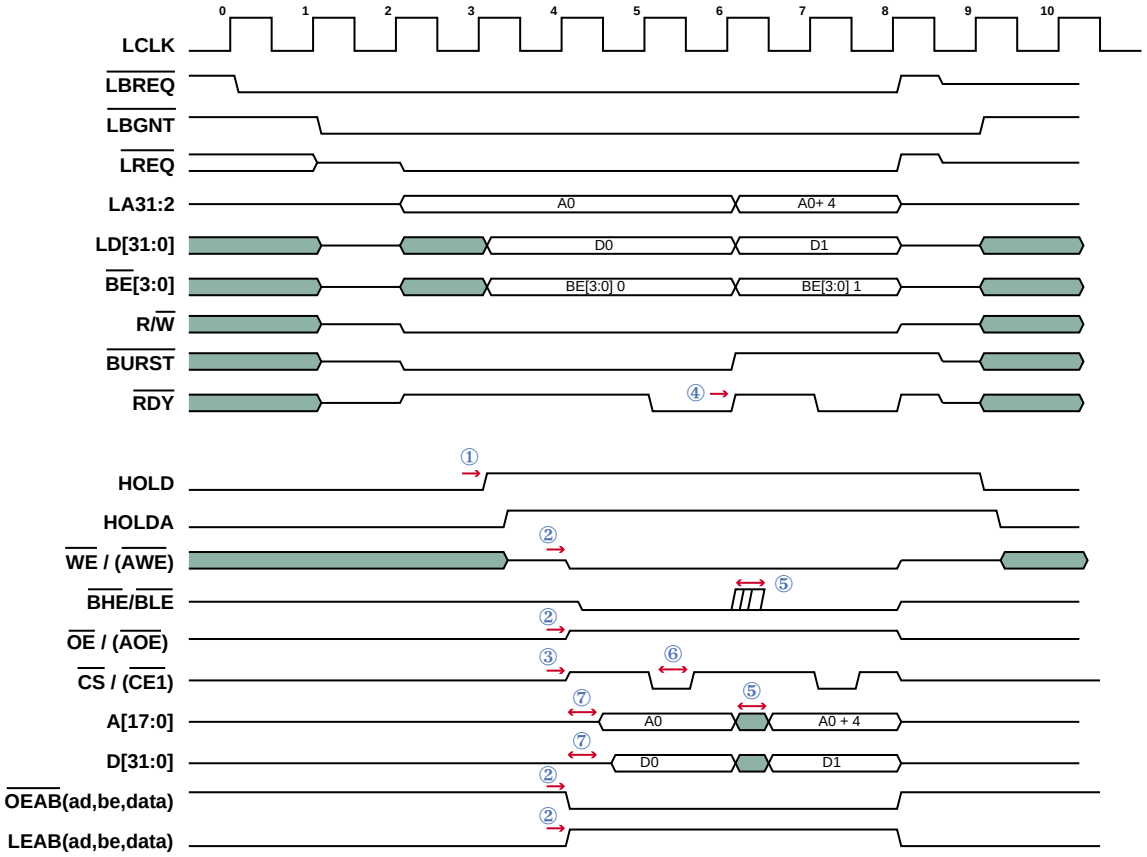
② At LCLK=4, the EPC has granted the external bus ($\overline{\text{HOLDA}}$ asserted low). Since it is a read access, the address byte enables are opened for the EPC-to-SRAM data path. The data buffers are opened for the SRAM-to-EPC data path. The $\overline{\text{WE}}$ signal is de-asserted high; the $\overline{\text{OE}}$ output enable is asserted low.

③ The maximum address access time (for IDT71V416 SRAM) is 15 ns. The delay time from the time the buffer is opened until the output is stable (for TI 18-bit universal transceiver bus SN54ABT16601, SN74ABT16601) is 6 ns max (worst case). The worst case scenario for the data to be ready is 21 ns plus the PLD's CLK-to-OUT (typically 5 ns), totalling 26 ns. Therefore, you should sample data on the rising edge of LCLK=6 (not LCLK=5), giving almost the entire cycle as margin. In an application where a slower LCLK is used (less than 40 MHz), you may strobe data on the rising edge of LCLK= 5. However it is better to have the cycle as margin in case of slower SRAMs, buffers, or PLD.

④ On LCLK=6 data is read to the EPC ($\overline{\text{RDY}}$ is asserted). On a burst, it takes on the worst case T_{COV} (EPC=15 ns max) delay time for the next address and byte enables to be stable. The delay time from change in the input to the output of the address and byte enable buffers is 5 ns maximum for TI 18-bit universal transceiver bus SN54ABT16601/SN74ABT16601. The maximum address access time for the IDT71V416 SRAM is 15 ns. Worse case is that it takes 35 ns for the subsequent data to be ready to strobe. Therefore, strobe data on the rising edge of LCLK=8, not LCLK=7, unless LCLK is less than 28 MHz.

4.2 EPC-to-SRAM Write Waveform

EPC to SRAM write cycle



EPC-to-SRAM Write Cycle Notes

① After LCLK=2, Where EPC has granted the BUS ($\overline{\text{LBGNT}}$ is asserted low), EPC starts the cycle by asserting LREQ and driving the address bus. On the rising edge of LCLK=3 the PLD decodes the address to HPI or SRAM access according to the aperture programming. When it decodes SRAM access, HOLD to EMIF control is asserted high. When the EMIF is ready to give up the external bus, it asserts back HOLDA asynchronous to LCLK.

② On LCLK=4, EPC has granted the external bus (HOLDA is asserted high). Since it is a write access, the address, byte enables, and data buffers are opened for EPC-to-SRAM data and address paths. The $\overline{\text{WE}}$ write signal is asserted low, OE output enable is de-asserted high (it is not a read cycle).

③ After LCLK=4, $\overline{\text{CE}}$ is sequentially driven high for one cycle and gate NOT with the LCLK in the next cycle. $\overline{\text{CE}}$ ends its sequential behavior when the access ends on LCLK=8. On LCLK=8, $\overline{\text{BURST}}$ is de-asserted high and $\overline{\text{RDY}}$ is asserted low, indicating the end of access.

④ On LCLK=6, data has already been written to SRAM, $\overline{\text{RDY}}$ is asserted in order to end the cycle for the next write cycle in a burst access or to end the access if it's not a burst. In this case it's a burst access, the address and the data of the next write cycle is driven by the EPC.

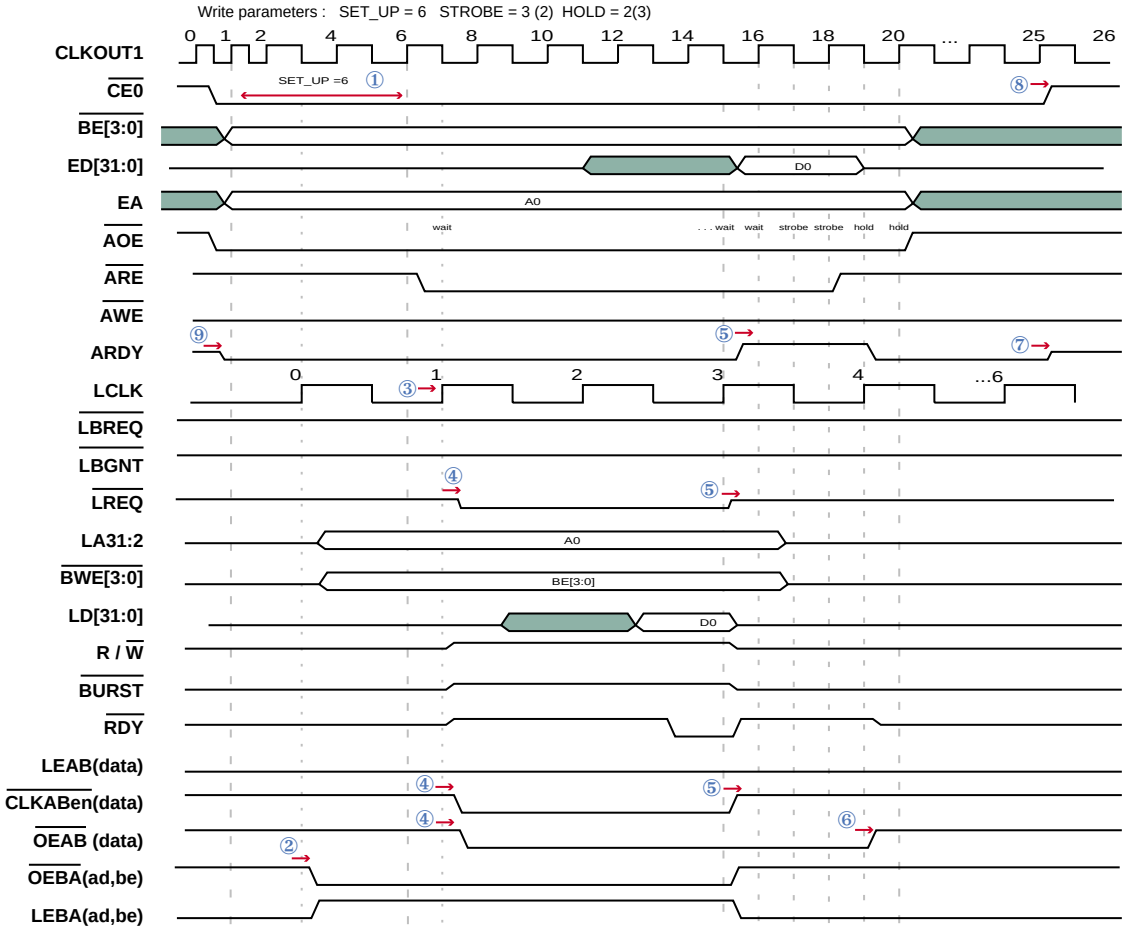
⑤ On a burst, it takes (worst case) Tcov (EPC=15 ns) delay time for the next address, byte enables, and the data to be stable. The delay time from change in the input to output valid for the address, byte enables and data buffers are 5 ns. Before asserting $\overline{\text{CS}}$ for the next write strobe, a set up of 20 ns is required. The EPC-to-SRAM waveform shows 1 LCLK inserted before asserting $\overline{\text{CS}}$ (20 ns at 50 MHz).

⑥ The $\overline{\text{CE}}$ STROBE period must be 8–10 ns minimum (following IDT71V416 SRAM data sheet). $^1/_2$ LCLK accomplishes the task.

⑦ The max delay time for the address, byte enables and the data on the first write cycle is T_{COV}(pld) + Tpd. Control signals-to-output valid for the buffers is typically 11 ns.

4.3 EMIF-to-EPC Read Waveform

EMIF to EPC Read Cycle



EMIF-to-EPC Read Cycle Notes

① SETUP>= 2, SETUP must be bigger or equal to two: The read cycle demonstrates the max case for CLKOUT1 which is 200 MHz and the max case for local (EPC max local clk) clk which is 50 MHz. SETUP period must be at least 2 (SETUP=2). SETUP=2 would give ARDY, which is asynchronously enabled by $\overline{CE0}$,a reasonable delay time (10 ns) to be asserted low before the STROBE period starts. Also, maintain the same order of operation when using SETUP period less than 6, since SETUP period won't be detectable for sure. Meaning, after LCLK=0 open the address and byte enable buffers even though the $\overline{ARE}$ signal would be already asserted and on LCLK=1 start the cycle on EPC side by asserting $\overline{LREQ}$ and the other control signals. Don't shave off a cycle by opening the address,

byte enable buffers and $\overline{\text{LREQ}}$ on the same clock. Otherwise you would need to take into consideration the PLD clock to out, the buffers delay time and the EPC's address and byte enables setup time. On 50 MHz LCLK that is not feasible. Note: If you want to detect the SETUP period for sure choose SETUP=6 or bigger for the same frequencies.

② LCLK = 0: $\overline{\text{CE0}}$ is asserted. On the rising edge of LCLK 0 PLD detects EMIF is in the SET UP period for an EMIF to EPC access cycle or in the STROBE period if you choose 2<=SETUP<6 (less than six but bigger or equal to 2). If $\overline{\text{LBGNT}}$ is not asserted, meaning EPC has not granted the bus, EMIF can access the EPC bus by enabling the address and byte enables buffers. Otherwise, EMIF is held the in the STROBE period with wait states inserted and the address and byte enable buffers closed.

③ LCLK = 1: If $\overline{\text{LBGNT}}$ is not asserted, the EPC has not granted the bus, EMIF-to-EPC access can proceed. The address and byte enable buffer opens to access the EPC's address and byte enable buses (OEAB(ad,be)), LEAB(ad,be)). If the EPC is granted the bus, wait state cycles are inserted waiting for $\overline{\text{LBGNT}}$ to go idle (i.e., not asserted); EMIF remains in the STROBE period with ARDY low and the buffers closed. When $\overline{\text{LBGNT}}$ goes idle, maintain the same order of operation as step 1.

④ LCLK = 1: $\overline{\text{ARE}}$ is asserted, a read cycle has started on the EMIF side. A read cycle starts on the rising edge of LCLK=1 by asserting $\overline{\text{LREQ}}$. $\overline{\text{BURST}}$, R/$\overline{\text{W}}$. The data buffers also open to allow data path from EPC to EMIF data bus. Data is latched on the rising edge of each LCLK ($\overline{\text{CLKBAen}}$ is asserted). Note that if the address and the byte enables buffers were not opened (note 1,2) they could be opened also here after the rising edge of LCLK=1. In this case, the delay time for the address and the byte enables before the rising edge of lclk=2 is: T_{COV}(pld) + tpd max (ad and be buffers to output) + EPCtsu(ad,be) ~ 5 + 6 +13 =24 ns typically. still safe for 40MHz or less. For higher frequencies maintain the same order of operation as explained in (note 1) and that is the safest design.

⑤ LCLK= 3: Data is ready to read from the EPC ($\overline{\text{RDY}}$ is asserted) on the rising edge of LCLK=3. The data has been latched in the data buffer. Latching is disabled ($\overline{\text{CLKBAen}}$ is de-asserted) after this clk.The max data delay time from clk to output on the latching buffer (the data buffer) is 5.2 ns. The setup time for ARDY strobe is 5 ns. The PLD T_{COV} typically 5 ns. Program the strobe to 2, where the data is valid and enough set up time is provided for ARDY. LREQ is de-asserted as the cycle has ended on the EPC side. The address and the byte enable buffers are closed (OEAB,LEAB(ad,be)) after this clock cycle as well.

⑥ The data buffer output enable is de-asserted synchronously with LCLK=4 to disable buffering.

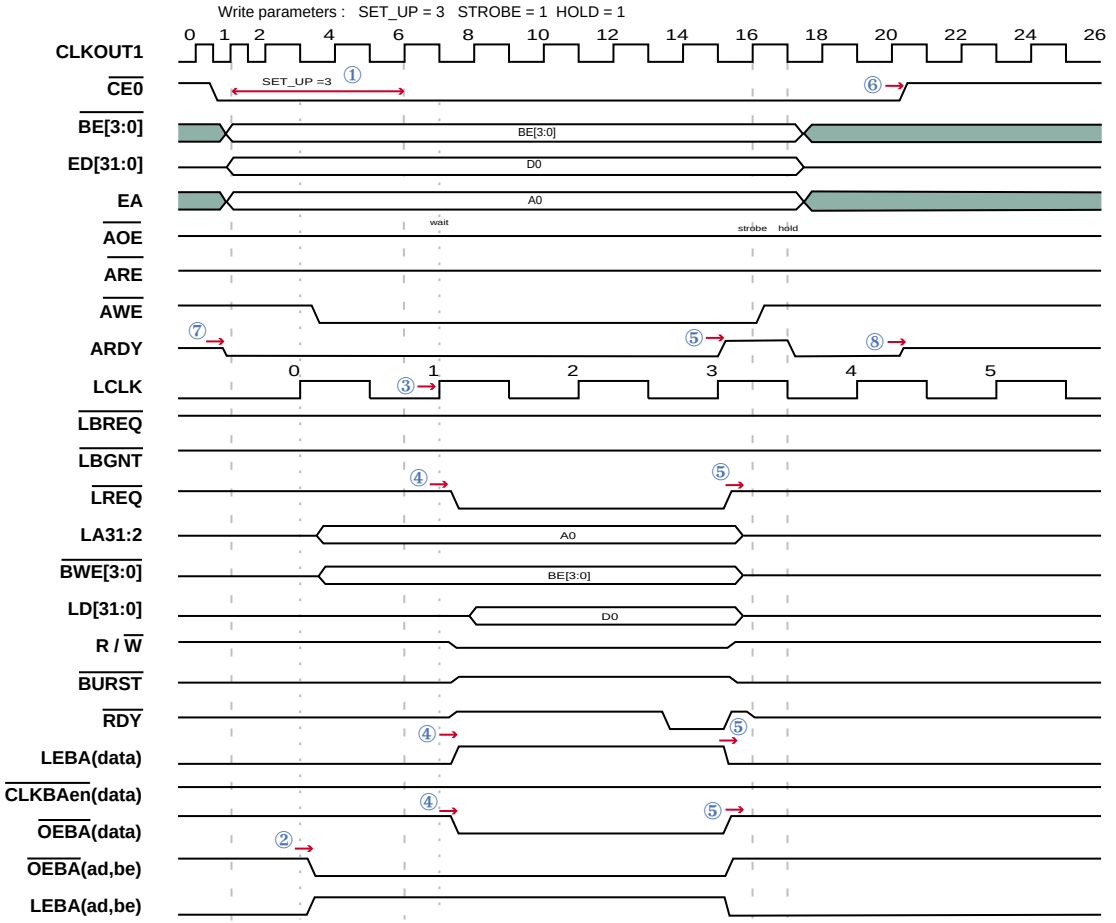
⑦ ARDY is tristated to high Z asynchronously with $\overline{\text{CE0}}$ de-assertion.

⑧ $\overline{\text{CE0}}$ (EMIF output) is de-asserted after seven minus the HOLD period clocks. in the Waveform HOLD=2, so the $\overline{\text{CE0}}$ is de-asserted on CLKOUT1 =25 (25-20=5), 7-2=5. After the HOLD period, if there is a back-to-back cycle, $\overline{\text{CE0}}$ is not de-asserted and the SETUP period for the next access starts immediately after HOLD.

⑨ ARDY is asynchronously asserted low with $\overline{\text{CE0}}$ assertion in order to prevent EMIF from proceeding in the STROBE period without getting synchronized with the EPC.

4.4 EMIF-to-EPC Write Waveform

EMIF to EPC Write Cycle



EMIF-to-EPC Write Cycle Notes

① SETUP>= 2, SETUP must be bigger or equal to two: The write cycle demonstrates the max case for CLKOUT1 which is 200 MHz and the max case for local (EPC max local clk) which is 50 MHz.

SETUP period just be at least 2 (SETUP=2). SETUP=2 would give ARDY, which is asynchronously out put enabled by $\overline{CE0}$, a reasonable delay time (10 ns) to be asserted low before the STROBE period starts. Also, maintain the same order of operation meaning after LCLK=0 open the address and byte enable buffers even though the $\overline{AWE}$ signal would be already asserted and on LCLK=1 start the cycle on EPC side by asserting $\overline{LREQ}$ and the other control

signal. Don't shave off a cycle by opening the address and byte enable buffers and $\overline{\text{LREQ}}$ on the same clock. Otherwise you would need to take into consideration the PLD clock to out, the buffers delay time and the EPC's address and byte enables setup time. On 50 MHz LCLK that is not feasible.

② LCLK = 0: $\overline{\text{CE0}}$ is asserted. On the rising edge of LCLK 0 PLD detects EMIF is in the SET UP period for an EMIF to EPC access cycle or in the STROBE period if SETUP is less than six. If $\overline{\text{LBGNT}}$ is not asserted, meaning EPC is not granted the bus, EMIF can access the EPC bus by enabling the address and byte enables buffers. Otherwise EMIF is held in the STROBE period with wait states inserted.

③ LCLK = 1: $\overline{\text{LBGNT}}$ is not asserted, meaning EPC has not granted the bus, EMIF to EPC access can proceed. The Address and byte enable buffers open to access EPC's address and byte enables buses. If the EPC is granted the bus, wait cycles are inserted waiting for $\overline{\text{LBGNT}}$ to go idle; EMIF remains in the STROBE period with ARDY low.

④ LCLK = 1: $\overline{\text{AWE}}$ is asserted, a write cycle starts on the rising edge of LCLK=1 by asserting $\overline{\text{LREQ}}$. $\overline{\text{BURST}}$, R/ $\overline{\text{W}}$. The data buffers are also opened to allow a data path from EMIF to the EPC data bus.

⑤ LCLK= 3: Data has been written to the EPC on the rising edge of LCLK=3. $\overline{\text{RDY}}$ was asserted low by the EPC. ARDY went high after the rising edge of LCLK=3 for half a clock. A single strobe is enough in the STROBE period to end the STROBE period. To the EPC, the end of the cycle is on the rising edge of LCLK=3. Data, address and the byte enable buffers are locked. The control signal to the EPC is tristated. For the EMIF, keep the following relationship for any design frequency or programming:

```
STROBE + HOLD + SETUP > TLCK
```

Example1: STROBE=1, HOLD=1, SETUP=3, (LCLK=50 MHz, CLKOUT1=200MHZ) (`5×5=25 > 20`)

Example2: STROBE=1, HOLD=1, SETUP=6, (LCLK=50 MHz, CLKOUT1=200MHZ) (`8×5=40 > 20`)

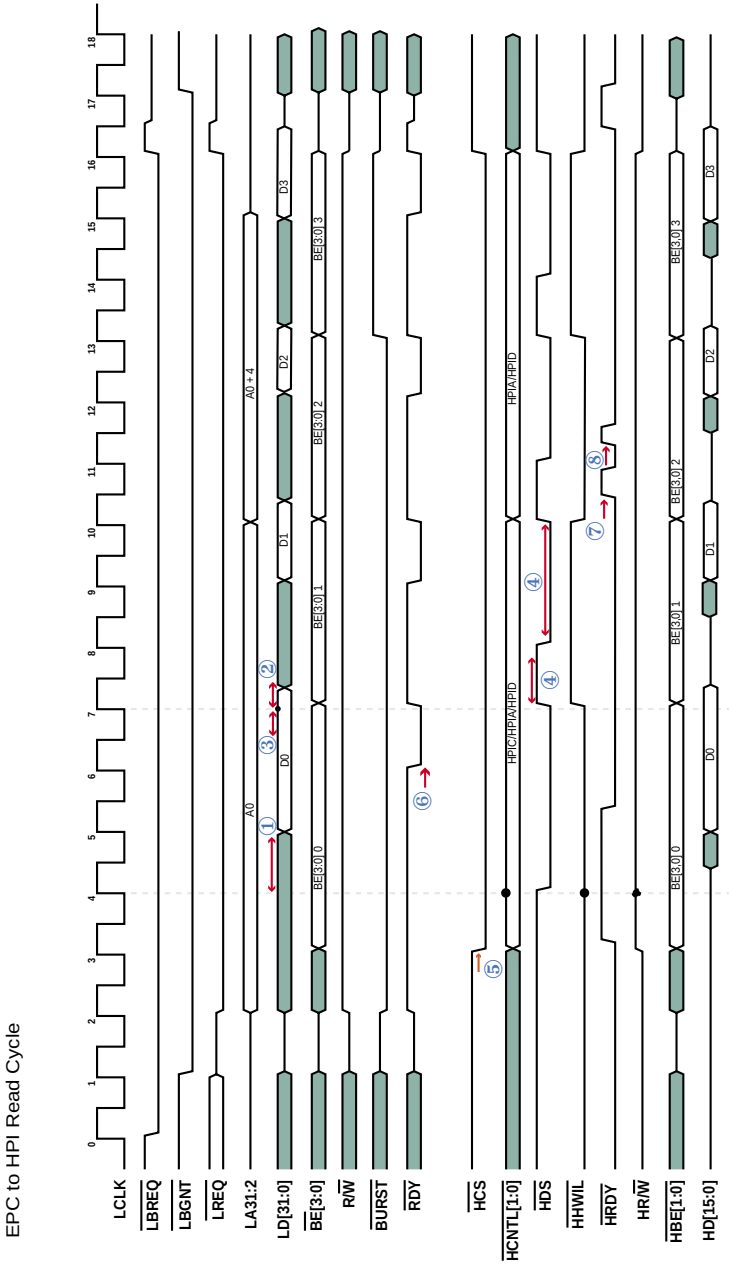
Note: $\overline{\text{RDY}}$ clock to out is 14 ns max (Tcov max 14 ns, see the EPC data sheet) plus the PLD delay enables us to use asynchronous ARDY assertion with $\overline{\text{RDY}}$ assertion on 50 MHz, Thus, ARDY is asserted high for $^1/_2$ clock after the cycle has ended on the EPC side.

⑥ $\overline{\text{CE0}}$ is de-asserted after 3 CLKOUT1 clocks from the end of the HOLD period (when HOLD is greater than 0 and 4 clocks if HOLD=0. In the waveform, HOLD=1, so the $\overline{\text{CE0}}$ is de-asserted on CLKOUT1 =20 (`20-17=3`), After the HOLD period, if there is a back to back cycle, $\overline{\text{CE0}}$ is not de-asserted and the SETUP period for the next access starts immediately after HOLD.

⑦ ARDY is asynchronously asserted low with $\overline{\text{CE0}}$ assertion in order to prevent the EMIF from proceeding in the STROBE period without getting synchronized with the EPC.

⑧ ARDY is tristated to high Z asynchronously with $\overline{\text{CE0}}$ de-assertion.

4.5 EPC-to-HPI Read Waveform



EPC-to-HPI Read Cycle Notes

① STROBE low to data valid is 12 ns minimum. This is a TI feature.

② DSP data HOLD time after STROBE high is 12 ns. This is a TI feature.

③ The EPC's data SETUP time is 8 ns. This is a V3 requirement.

④ STROBE (HSTRB) minimum assertion or de-assertion duration time is 2 CLKOUT1 cycles. At f= 200 MHz, CLKOUT1= 5 ns. The assertion or the de-assertion period in this design is 1 LCLK at least, which is 20 ns minimum. (f=50 MHz EPC max speed).

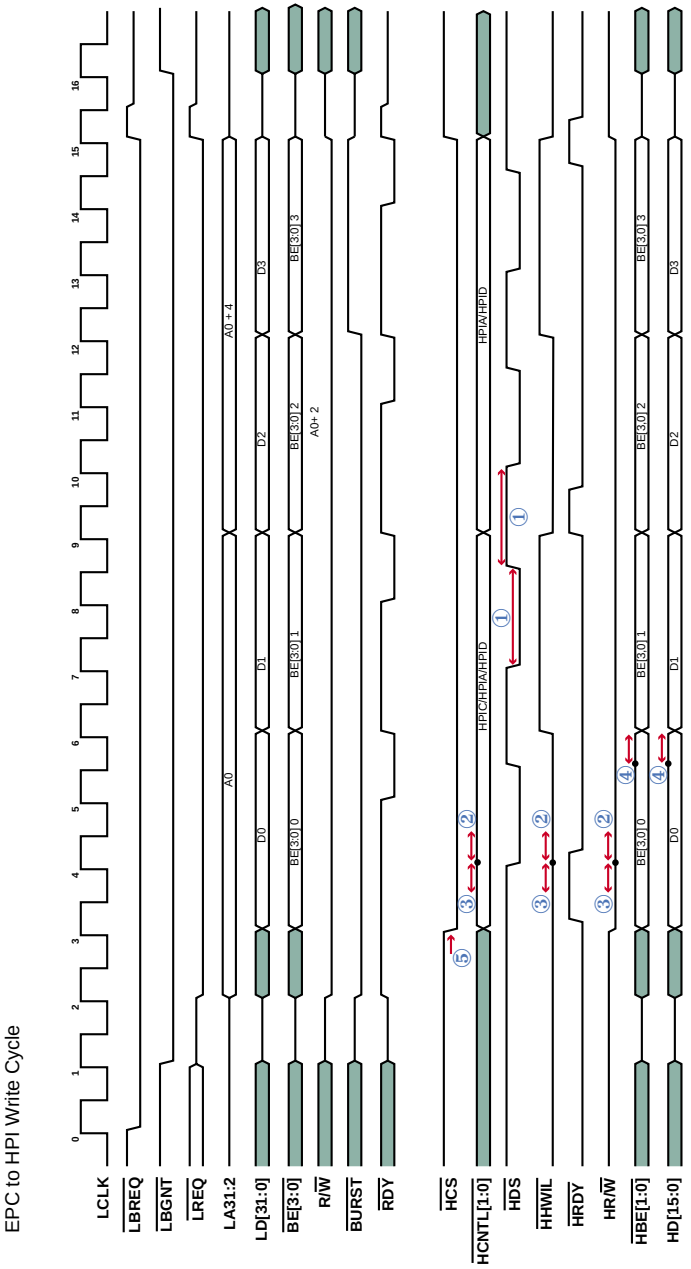
⑤ $\overline{\text{HCS}}$ HPI Chip select is asserted to start an access when the EPC has been given the bus ($\overline{\text{LBGNT}}$ is asserted) and the EPC has started the cycle ($\overline{\text{LREQ}}$ is asserted). Only a cycle after LREQ is asserted can the PLD determine if the access is to HPI or to SRAM. On LCLK=3 the access address is driven on LA[31:2] therefore PLD can decode the access to HPI.

⑥ On LCLK=6 $\overline{\text{HRDY}}$ is asserted therefor we can assert $\overline{\text{RDY}}$ for EPC to read the data on LCLK=7. On LCLK=7 $\overline{\text{RDY}}$ is asserted: data is read to the EPC, and $\overline{\text{RDY}}$ and $\overline{\text{HDS}}$ signals are de-asserted to end the current read cycle. There is no race condition between $\overline{\text{HDS}}$ and $\overline{\text{RDY}}$ so PLD delay time is more than enough hold time for the data.

⑦ $\overline{\text{HRDY}}$ might be de-asserted on the de-assertion of $\overline{\text{HDS}}$ after the transfer of second halfword.

⑧ $\overline{\text{HRDY}}$ might be de-asserted on the assertion of $\overline{\text{HDS}}$ in the beginning of a read or a write cycle (the first halfword).

4.6 EPC-to-HPI Write Waveform



EPC-to-HPI Write Cycle Notes

① STROBE (HSTRB) minimum assertion or de-assertion duration time is 2 CLKOUT1 cycles.

At f= 200 MHz, CLKOUT1= 5 ns. The assertion and the de-assertion of $\overline{\text{HDS}}$ is 1 LCLK which is 20 ns minimum. (f=50 MHz EPC max speed)

② The HOLD time after strobe low is 2 ns. The signals strobe on the falling edge of $\overline{\text{HDS}}$ are HCNTL, HHWIL, HR/W.

③ SETUP time for the signals strobe is1 ns from the falling edge of $\overline{\text{HDS}}$.

④ Data and byte enables strobe is on the rising edge of $\overline{\text{HDS}}$. The HOLD time required is 1 ns. $\overline{\text{HDS}}$ is negatively gated with LCLK=5 so the rising edge of $\overline{\text{HDS}}$ is on the falling edge of LCLK= 5.5 which provides $^1/_2$ LCLK less T_{COV}(pld) HOLD time.

⑤ $\overline{\text{HCS}}$ HPI Chip select is asserted to start an access when the EPC has been given the bus ($\overline{\text{LBGNT}}$ is asserted) and the EPC has started the cycle ($\overline{\text{LREQ}}$ is asserted). Only a cycle after $\overline{\text{LREQ}}$ is asserted can the PLD determine if the access is to HPI or to SRAM. On LCLK=3 the access address is driven on LA[31:2] therefore PLD can decode the access to the HPI.

5.0 Programming

Programming the EPC and TMS320C6 involves programming a number of registers. This chapter describes only those which have a direct effect on the hardware design of this application.

The V360EPC has two programmable apertures to access the local bus from PCI, and two programmable apertures to access PCI from the local bus. This application accesses two peripherals on the local bus from PCI: the HPI (the DSP's Host Processor Interface) and SRAM memory. Other memory peripherals residing on the EMIF bus, like SDRAM or SBSRAM, may be accessed from PCI. This application uses the asynchronous SRAM interface since it is the most common application. For SRAM accesses, define one PCI-to-Local aperture for HPI and the other as a PCI-to-Local aperture. Both apertures are identical—there is no advantage to using one or the other.

1. PCI accesses HPI through aperture 1.

2. PCI accesses SRAM through aperture 0.

Bit 30 on the local bus determines which aperture is accessed. See Table 6: Address Decoding on page 26.

5.1 PCI to HPI

Maintain the EPC burst for best performance and maximum utilization of the local bus when EPC is accessing HPI. Use the following burst format:

HPIC HPIA HPID HPID.HPID up to 1Mbyte data

Where HPIC is the control register, HPIA is the address register, and HPID is the data register.

This access format is possible if you use two sequential addresses of HPIC, HPIA, and then continue the sequential addresses up to 1 Mbit (125 KB) of HPID by using the DSP's address increment feature. In total, 128 KB of sequential data address are needed following the HPIC and HPIA addresses. You can also access 128 KB of the processor internal memory without address increments in a sustain burst fashion. Single cycles are always able to access any of the registers.

In order to achieve full performance of the DSP as described above, use the 1 MB aperture size. The 1 MB aperture size is the minimum available. Use 256 KB for incremented or non-incremented addresses, and 2 addresses for HPIC and HPIA registers (512 KB—8 bytes is be used).

Table 6: Address Decoding describes how to design the system:

Table 6: Address Decoding

A30[a]	A19	A18	A3	A2	Register	Aperture
0	0	x	x	0	HPIC	Aperture 1
0	0	x	x	1	HPIA	Aperture 1
0	0	x	x	0	HPIC	Aperture 1
0	0	x	x	1	HPIA	Aperture 1
0	1	0	x	x	HPID with increment [b]	Aperture 1
0	1	1	x	x	HPID with no increment[c]	Aperture 1
1	x	x	x	x	Z	Aperture 0

a. You may select any bit from A20 to A31 to determine accesses to HPI or SRAM. Choosing A30 provides maximum flexibility if a 1 GB SRAM aperture is to be used (see the Aperture Programming Example on page 27).

b. For best performance with longer burst accesses, the addresses of HPIC, HPIA and HPID should be sequential, especially when using the following address increments:
0xX7FFF8 for the HPIC address; (X = bits 31–20 in the address bus)
0xX7FFFC for the HPIA address;
0xX80000 for the HPID first address up to 1 MB data burst access.
The EPC max burst size is 1 KB, burst accesses more than 1 KB is broken into 1 KB bursts. If the DMA is set to transfer 128 KB + 8 bytes (or less) of address, control, and data starting from HPIC, the EPC breaks the transfers into 1 KB bursts.

c. 0xXC0000 is the first address in 1 Mb sequential addresses for non-incremental access. HPIC and HPIA can be any combination of addresses A19, A18="00", or A18="01". Accessing HPIC and HPIA can be a burst of two if the addresses are sequential. To access HPID for up to 1 Mbit burst size is possible in a separate access—the EPC breaks the transfer into 1 KB bursts. From a system level, a programmer can set the EPC's DMA to transfer any amount of data up to 1 MB.

The access to the HPI interface registers is determined by HNCTL[1:0] pins according to the following table.

Table 7: HPI Interface

HCNTL1	HCNTL0	Description
0	0	Access to HPIC
0	1	Access to HPIA
1	0	Access to HPID with HPIA increment
1	1	Access with HPID, HPIA is not affected

A30 determines whether the access is to HPI or SRAM. A19, A18, and A2 determine whether the access is to HPIC, HPIA, HPID with increment, or HPID with no increment.

NOTE: Program BE_OMODE ='1' in LB_CFG register for normal 292 mode operation.

Aperture Programming Example

PCI_BASE1 = 0x5A500008

Prefetch is on.

PCI_MAP1 = 0xA5A00003

Aperure1 size is 1 MB; aperture and register are enabled.

PCI_BASE0 = 0x80000008

PCI_MAP0 = 0x400000A3 (or, since we are using A30 to decode access, we may also use 0xC00000A3; the A31 value is "don't care.")

Aperure0 size is 1 GB; aperture and register are enabled.

LB_SIZE = 0x00300000

We have programmed the region to be 16-bit unpacked. The reason it should be unpacked is that accesses must be comprised of two 16-bit accesses, the higher bytes and the lower bytes in a word. This is true even if one half of the word (either the higher 16-bit or the lower 16-bit) has no data to deliver. The internal data boundary is 32 bits (one word), so the first access is the higher or the upper half of the word (upper or lower is programmable in the DSP). In packed mode, the halfword, which does not have data, is deleted and another halfword (from the next word) replaces it. This packing feature uses the maximum 16-bit bus bandwidth; however we cannot use this feature in this design because the DSP internal bus has a 32-bit, not a 16-bit, word boundary.

5.2 PCI to SRAM

Aperture 0 is dedicated to PCI to SRAM accesses. There are no EPC programming issues that are not mentioned in Section 5.1—PCI to HPI.

5.3 EMIF to PCI

For EMIF-to-PCI accesses, $\overline{CE0}$ is mapped for EMIF-to-PCI memory space and $\overline{CE1}$ is mapped to SRAM memory space. When EMIF wants to access PCI, it drives $\overline{CE0}$ low and drives the address bus EA[21:2]. The EMIF address bus EA[21:2] is connected to LA[21:2] through the address buffers. The EPC's LA[22:31] higher address bits are not being used when the EPC is the target, so pull these bits up.

With the pull-ups in place, address decoding of **LB_BASE0** or **LB_BASE1** depends on bit 21. The address decoding uses a minimum aperture size of 1 MB, which decodes the address bits 31:20. Bits 31:22 are pulled high leaving bit 21 for decoding a 1 MB aperture. If you need a 2 MB aperture, enable bit A30 as an output bit instead of pulling it high. Then drive it when the PLD decodes an access to the EPC ($\overline{CE0}$ is being driven). This application uses a 1 MB Local-to-PCI aperture size, **LB_BASEn**, so LA[31:22] is pulled up. When EA[21]=1, the EPC decodes an access to PCI through the aperture **LB_BASE0.**

LB_BASE0 = 0xFFF00000

6.0 PLD Source Code

The following source code is written in VHDL.

6.1 VHDL Conventions

There are some conventions that have been used in writing the VHDL equations that help to make them more readable:

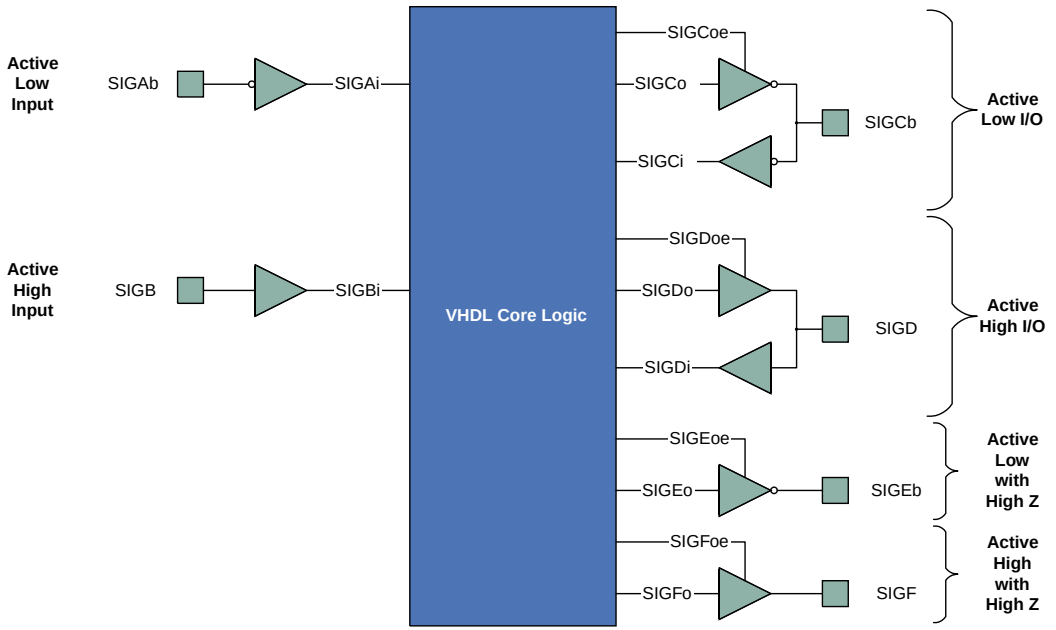
Signal names are all in upper case except as outlined below.

Signals that are active low (TSb, TAb, BWEb, BURSTb) are indicated by a lower case 'b' at the end of the signal name. All active low signals are converted to active high when used in the logic generation portion of the VHDL code. Active low signals are indicated by an overline ($\overline{\text{LBGNT}}$) throughout the rest of this document.

Top level signals pass through an I/O buffer stage as shown in Figure 4: Signal Names in Relation to I/O Buffering. This converts everything to positive true logic and also converts weak input highs/lows 'H'/'L' to'1'/'0'.

All VHDL core logic is written in positive true logic. That is, an active low signal (such as $\overline{\text{TS}}$, called TSb in VHDL) is converted to an active high equivalent (as in Figure 4) and then used by the core logic in the active high sense (TSi ='1' means $\overline{\text{TS}}$ is active).

Figure 4: Signal Names in Relation to I/O Buffering



6.2 VHDL Code

```
library IEEE;
use IEEE.std_logic_1164.all;

--=============================================================================================
--The objective of this VHDL code is to interface TI TMS320C6x DSP to PCI bus via V3 Semiconductor
V360EPC PCI bridge following the “High Performance Design” shown in Figure 1.
--=============================================================================================
entity ti_epc is
    port(
---------------------
-- Local clock
---------------------
        CLK          :in     std_logic;
---------------------
-- EPC interface
---------------------
        RESETb       :in     std_logic;
        LBREQb       :in     std_logic;
        LBGNTb       :out    std_logic;
        LREQb        :inout  std_logic := 'Z';
        RDYb         :inout  std_logic := 'Z';
        BURSTb       :inout  std_logic := 'Z';
        EPC_RWb      :inout  std_logic := 'Z';
        A2           :in     std_logic;
        A18          :in     std_logic;
        A19          :in     std_logic;
        A30          :in     std_logic;
---------------------
-- HPI interface
---------------------
        HCNTL        :out    std_logic_vector(1 downto 0);
--      HPI_RWb      :out    std_logic;--can be directly connected
        HDSb         :out    std_logic;
        HCSb         :out    std_logic;
        HRDYb        :in     std_logic;
---------------------
--EMIF interface
---------------------
        EMIF_HOLDb   :out    std_logic;
        EMIF_HOLDAb  :in     std_logic;
        CE0b         :in     std_logic;
        CE1b         :inout  std_logic := 'Z';
        ARDY         :out    std_logic;
        AOEb         :inout  std_logic := 'Z';
        AWEb         :inout  std_logic := 'Z';
        AREb         :in     std_logic;
-------------------------------------------------
--NOTE:
--Buffers direction AB=EPC to EMIF; BA=EMIF to EPC
-------------------------------------------------
-------------------------------------------------
--BA= emif to epc writes or epc to sram reads
-------------------------------------------------
        LEBA_DATA    :out    std_logic;
        OEBA_DATAb   :out    std_logic;
-----------------------------------------------------------------------------------------------
--for EMIF to EPC reads (latching register is included in this case only) and EPC to SRAM writes
-----------------------------------------------------------------------------------------------
        CLKABEN_DATAb:out    std_logic;
        LEAB_DATA    :out    std_logic;
```



```
        OEAB_DATAb   :out    std_logic;

        LEAB_AD_BE   :out    std_logic;
        OEAB_AD_BEb  :out    std_logic;

        LEBA_AD_BE   :out    std_logic;
        OEBA_AD_BEb  :out    std_logic

 );
end ti_epc;

-------------------------------------------------
---------      Signal Declaration       ---------
-------------------------------------------------

---------------------
-- EPC interface
---------------------

--Inputs
        signalRESETi                 : std_ulogic;
        signal  LBREQi               : std_ulogic;
        signalLREQi                  : std_ulogic;
        signal  RDYi                 : std_ulogic;
        signal  BURSTi               : std_ulogic;
        signal  EPC_RWi              : std_ulogic;
        signal  A2i,A18i,A19i,A30i   : std_ulogic;
--Outputs
        signal  LBGNTo               : std_ulogic;

        signal   LREQo               : std_ulogic;
        signal   LREQoe              : std_ulogic;
        signal   BURSTo              : std_ulogic;
        signal   BURSToe             : std_ulogic;
        signal   EPC_RWo             : std_ulogic;
        signal   EPC_RWoe            : std_ulogic;
        signal   RDYo                : std_ulogic;
        signal   RDYoe               : std_ulogic;
---------------------
-- HPI interface
---------------------
        signal   HRDYi               :std_ulogic;
        signal   HCNTLo              :std_ulogic_vector(1 downto 0);
        signal   HCNTLoe             :std_ulogic;
--      signal   HPI_RWo             :std_ulogic; --Directly connected
--      signal   HPI_RWoe            :std_ulogic;
        signal   HDSo                :std_ulogic;
        signal   HCSo                :std_ulogic;
---------------------
--EMIF interface
---------------------
--Inputs
        signal EMIF_HOLDAi       :std_ulogic;
        signal CE0i              :std_ulogic;
        signal AREi              :std_ulogic;
        signal CE1i              :std_ulogic;
        signal AOEi              :std_ulogic;
        signal AWEi              :std_ulogic;
--Outputs
        signal EMIF_HOLDo        :std_ulogic;

        signal ARDYo             :std_ulogic;
        signal ARDYoe            :std_ulogic;
```

```
---------------------
--SRAM Interface
---------------------
        signal CE1o             :std_ulogic;
        signal CE1oe            :std_ulogic;
        signal AOEo             :std_ulogic;
        signal AOEoe            :std_ulogic;
        signal AWEo             :std_ulogic;
        signal AWEoe            :std_ulogic;
---------------------
--Buffers Interface
---------------------

        signal LEBA_DATAo       :std_ulogic;
        signal OEBA_DATAo       :std_ulogic;
        signal CLKABEN_DATAo    :std_ulogic;
        signal LEAB_DATAo       :std_ulogic;
        signal OEAB_DATAo       :std_ulogic;
        signal LEAB_AD_BEo      :std_ulogic;
        signal OEAB_AD_BEo      :std_ulogic;

        signal LEBA_AD_BEo      :std_ulogic;
        signal OEBA_AD_BEo      :std_ulogic;

-----------------------------------
--EPC to SRAM state machine signals
-----------------------------------
        signal ARB_CS                   :std_ulogic;
        signal EPC_SRAM_CS,EPC_SRAM_NS  :std_ulogic_vector(2 downto 0) ;
        constant EPC_SRAM_IDLE          :std_ulogic_vector(2 downto 0):="000";
        constant EPC_SRAM_ARB           :std_ulogic_vector(2 downto 0):="001";
        constant EPC_SRAM_WAIT          :std_ulogic_vector(2 downto 0):="011";
        constant EPC_SRAM_STROBE        :std_ulogic_vector(2 downto 0):="110";
-----------------------------------
--EMIF to EPC state machine signal
-----------------------------------
        signal EMIF_EPC_CS,EMIF_EPC_NS  :std_ulogic_vector(3 downto 0);
        constant EMIF_EPC_IDLE          :std_ulogic_vector(3 downto 0):="0000";
        constant EMIF_EPC_OPEN_BUF      :std_ulogic_vector(3 downto 0):="0001";
        constant EMIF_EPC_STARTCY       :std_ulogic_vector(3 downto 0):="0101";
        constant EMIF_EPC_WAIT4RDY      :std_ulogic_vector(3 downto 0):="0111";
        constant EMIF_EPC_STROBE        :std_ulogic_vector(3 downto 0):="1000";
        constant EMIF_EPC_WAIT4B2B      :std_ulogic_vector(3 downto 0):="0010";
-----------------------------------
-- EPC to HPI state machine signals
-----------------------------------
        signal EPC_HPI_CS,EPC_HPI_NS     :std_ulogic_vector(2 downto 0);
        signal NOT_LAST                  :std_ulogic;
        signal MASK                      :std_ulogic;
        signal MASK_PAR                  :std_ulogic;
        constant EPC_HPI_IDLE            :std_ulogic_vector(2 downto 0):="000";
        constant EPC_HPI_STARTCY         :std_ulogic_vector(2 downto 0):="001";
        constant EPC_HPI_STROBE1         :std_ulogic_vector(2 downto 0):="011";
        constant EPC_HPI_STROBE2         :std_ulogic_vector(2 downto 0):="010";
        constant EPC_HPI_DEASTROB        :std_ulogic_vector(2 downto 0):="100";



--      type EMIF_EPC_STATE is (EMIF_EPC_IDLE,
EMIF_EPC_OPEN_BUF,EMIF_EPC_WAIT4RDY,EMIF_EPC_STROBE,EMIF_EPC_B2B);

begin
```



```
--=============================================================================
-- I/O Buffers: Take the external I/O and create active high signals for internal use
--=============================================================================

----------------------
--EPC_Interface_Inputs
----------------------
        RESETi      <= not to_x01(RESETb);
        LBREQi      <= not to_x01(LBREQb);
        LREQi       <= not to_x01(LREQb);
        RDYi        <= not to_x01(RDYb);
        BURSTi      <= not to_x01(BURSTb);
        EPC_RWi     <= to_x01(EPC_RWb);
        A2i         <= to_x01(A2);
        A18i        <= to_x01(A18);
        A19i        <= to_x01(A19);
        A30i        <= to_x01(A30);
----------------------
--EPC_Interface_Outputs
----------------------
        LBGNTb      <= not to_x01(LBGNTo);
        LREQb       <= not to_x01(LREQo) when (LREQoe='1') else 'Z' ;
        EPC_RWb     <= to_x01(EPC_RWo) when (EPC_RWoe='1') else 'Z';
        RDYb        <= not to_x01(RDYo) when (RDYoe='1') else 'Z' ;
        BURSTb      <= not to_x01(BURSTo) when (BURSToe='1') else 'Z';

----------------------
--HPI_interface_Inputs
----------------------
        HCNTL       <= To_X01(To_StdlogicVector(HCNTLo)) when (HCNTLoe='1');
--      HPI_RWb     <= not to_x01(HPI_RWo) when (HPI_RWoe='1');
        HDSb        <= not to_x01(HDSo);
        HCSb        <= not to_x01(HCSo);
        HRDYi       <= not to_x01(HRDYb);

----------------------
--EPC to SRAM access)
----------------------
        EMIF_HOLDAi <= not to_x01(EMIF_HOLDAb);
        CE0i        <= not to_x01(CE0b);
        CE1i        <= not to_x01(CE1b);
        AOEi        <= not to_x01(AOEb);
        AWEi        <= not to_x01(AWEb);
        AREi        <= not to_x01(AREb);
----------------------
--EMIF_Interface_Outputs
----------------------
        EMIF_HOLDb  <= not to_x01(EMIF_HOLDo);
        CE1b        <= not to_x01(CE1o) when (CE1oe='1') else 'Z';
        ARDY        <= to_x01(ARDYo) when (ARDYoe='1');
        AOEb        <= not to_x01(AOEo) when (AOEoe='1') else 'Z';
        AWEb        <= not to_x01(AWEo) when (AWEoe='1') else 'Z';
----------------------
--Buffers Interface
----------------------
        LEBA_DATA         <= to_x01(LEBA_DATAo);
        OEBA_DATAb        <= not to_x01(OEBA_DATAo);
        CLKABEN_DATAb     <= not to_x01(CLKABEN_DATAo);
        LEAB_DATA         <= to_x01(LEAB_DATAo) ;
        OEAB_DATAb        <= not to_x01(OEAB_DATAo);
        LEAB_AD_BE        <= to_x01(LEAB_AD_BEo) ;
        OEAB_AD_BEb       <= not to_x01(OEAB_AD_BEo);
```

```
        LEBA_AD_BE        <= to_x01(LEBA_AD_BEo);
        OEBA_AD_BEb       <= not to_x01(OEBA_AD_BEo);
-------------------------------------------------


--#############################################################################--
--#############################################################################--
--###                                                                       ###--
--###         This is the core logic using all active high signals          ###--
--###                                                                       ###--
--#############################################################################--
--#############################################################################--
----------------------
--ARB process
----------------------
    LBGNTo   <= ARB_CS;
----------------------
--Buffers control (EMIF to EPC buffers direction and isolation control)
----------------------
  OEAB_AD_BEo    <= EPC_SRAM_CS(1);                    -- EPC to SRAM
  LEAB_AD_BEo    <= EPC_SRAM_CS(1);                    -- EPC to SRAM
  OEBA_AD_BEo    <= EMIF_EPC_CS(0);                    -- EMIF to EPC read and write
  LEBA_AD_BEo    <= EMIF_EPC_CS(0);                    -- EMIF to EPC read and write
  OEAB_DATAo     <= (EPC_SRAM_CS(1) and not(EPC_RWi))  -- EPC to SRAM AB direction on writes only
                    or (AREi and AOEi and CE0i);       -- EMIF to EPC read
  LEAB_DATAo     <= (EPC_SRAM_CS(1) and not(EPC_RWi)); -- EPC to SRAM AB on writes only
  CLKABEN_DATAo  <= EMIF_EPC_CS(2) and EPC_RWo;        -- EMIF to EPC read clocking latch
  OEBA_DATAo     <= (EPC_SRAM_CS(1) and EPC_RWi)       -- EPC to SRAM
                    or (EMIF_EPC_CS(2) and EPC_RWo);   -- EMIF to EPC write
  LEBA_DATAo     <= (EPC_SRAM_CS(1) and (not EPC_RWi)) -- EPC to SRAM
                    or (EMIF_EPC_CS(2) and (not EPC_RWo)); -- EMIF to EPC write
-------------------------------
-- Arbitration process. EMIF can always access EPC's local bus.
--If EMIF does not need to access EPC's local bus and EPC needs to access local bus
--for HPI or SRAM accesses the arbiter grants the EPC the local bus.
-------------------------------
ARB: process(CLK, RESETi)
    variable ARB_NS : std_ulogic;
    begin
        if (RESETi='1') then
            ARB_CS <= '0';
        elsif (CLK'event and (CLK='1')) then
            case ARB_CS is
                when '0' =>
                    if ((LBREQi and not CE0i)='1') then
                        ARB_NS :='1';
                    else
                        ARB_NS :='0';
                    end if;
                when '1' =>
                    if (LBREQi='0') then
                        ARB_NS :='0';
                    else
                        ARB_NS:='1';
                    end if;
                when others =>
                    ARB_NS:='0';
                end case;
        end if;
        ARB_CS <= ARB_NS;
    end process ARB;

-------------------------------------------------------------
```



```
-- Update EPC to SRAM, EPC to HPI and EMIF to EPC state machines
-----------------------------------------------------------------
UPDATE_STATES: process(CLK,RESETi)
    begin
        if (RESETi='1') then
            EPC_SRAM_CS <= EPC_SRAM_IDLE;
            EMIF_EPC_CS <= EMIF_EPC_IDLE;
            EPC_HPI_CS  <= EPC_HPI_IDLE;
        elsif (CLK'event and (CLK='1')) then
            EPC_SRAM_CS <= EPC_SRAM_NS;
            EMIF_EPC_CS <= EMIF_EPC_NS;
            EPC_HPI_CS  <= EPC_HPI_NS;
        end if;
    end process UPDATE_STATES;

--------------------------------
--        EPC to SRAM         --
--------------------------------

    EMIF_HOLDo    <= EPC_SRAM_CS(0) or EPC_SRAM_CS(1) or EPC_SRAM_CS(2);

    CE1o          <= EPC_SRAM_CS(2) and MASK;      -- keep asserting CE1 for reads, strobe for a
                                                      half a clock for writes.
    CE1oe         <= EMIF_HOLDo and EMIF_HOLDAi;
    AOEoe         <= EPC_SRAM_CS(1);
    AWEoe         <= EPC_SRAM_CS(1);
    RDYo          <= (EPC_SRAM_CS(2) or                              --EPC to SRAM
                      (EPC_HPI_CS(1) and (not EPC_HPI_CS(0))));      --strobe2 state EPC to HPI
    RDYoe         <= LBGNTo;

LACH: process(EPC_RWi, EMIF_HOLDAi)
        begin
            if (EMIF_HOLDAi ='0') then
                AOEo <= EPC_RWi;
                AWEo <= not EPC_RWi;
            end if;
        end process LACH;

EPC_SRAM_STM: process(EPC_SRAM_CS,LBREQi,LBGNTo,LREQi,EMIF_HOLDo,EMIF_HOLDAi,BURSTi,A30i)
-- assert RDY for EPC and strobe data on first half cycle on a write
--                                                  |EPC to SRAM in process-open buffers and OEs
--                                                  ||HOLD EMIF from driving it's external bus.
                                                    ||            EPC is requesting the bus
--                                                  |||
--      constant EPC_SRAM_IDLE    :std_logic_vector:="000";
--      constant EPC_SRAM_ARB     :std_logic_vector:="001";         -- ask for EMIF bus (assert
                                                                        HOLD and wait for HOLDA)
--      constant EPC_SRAM_WAIT    :std_logic_vector:="011";
--      constant EPC_SRAM_STROBE  :std_logic_vector:="110";
--      constant EPC_SRAM_END     :std_logic_vector:="010";


    begin
        case EPC_SRAM_CS is
            when EPC_SRAM_IDLE =>
                if ((LBREQi and LBGNTo and LREQi and A30i)= '1') then --Access to SRAM
                    EPC_SRAM_NS <= EPC_SRAM_ARB;
                else
                    EPC_SRAM_NS <= EPC_SRAM_IDLE; -- wait until LREQ is asseted to start cycle .
                end if;
            when EPC_SRAM_ARB =>
                if (EMIF_HOLDAi='1') then
                    EPC_SRAM_NS <= EPC_SRAM_WAIT;
```

```
                else
                    EPC_SRAM_NS <= EPC_SRAM_ARB;
                end if;
            when EPC_SRAM_WAIT =>
                    EPC_SRAM_NS <= EPC_SRAM_STROBE;
            when EPC_SRAM_STROBE =>
                if (BURSTi='1') then
                    EPC_SRAM_NS <= EPC_SRAM_WAIT;
                else
                    EPC_SRAM_NS <= EPC_SRAM_IDLE;
                end if;
--          when EPC_SRAM_END =>
--                  EPC_SRAM_NS <= EPC_SRAM_IDLE;
            when others =>
                EPC_SRAM_NS <= EPC_SRAM_IDLE;
        end case;
    end process EPC_SRAM_STM;

--------------------------------
-- EMIF to EPC                 --
--------------------------------
    LREQo         <= EMIF_EPC_CS(2);
    LREQoe        <= not LBGNTo;
    BURSTo        <= '0';
    BURSToe       <= not LBGNTo;
    EPC_RWoe      <= not LBGNTo;
    ARDYo         <= EMIF_EPC_CS(3);
    ARDYoe        <= CE0i;
EMIF_EPC_STM : process(EMIF_EPC_CS, AREi,AWEi,CE0i, RDYi,LBGNTo)

-- ARDY
--                                                          |LREQ,BURST,R/W,buffers enable
--                                                          ||Wait for ready during the cycle or
                                                                check if there is back to back
                                                                access after the end of the
                                                                cycle
--                                                          |||ADD,BE buffer enable
--                                                          ||||
--      constant EMIF_EPC_IDLE          :std_ulogic_vector:="0000";
--      constant EMIF_EPC_OPEN_BUF      :std_ulogic_vector:="0001";
--      constant EMIF_EPC_STARTCY       :std_ulogic_vector:="0101";
--      constant EMIF_EPC_WAIT4RDY      :std_ulogic_vector:="0111";
--      constant EMIF_EPC_STROBE        :std_ulogic_vector:="1000";
--      constant EMIF_EPC_WAIT4B2B      :std_ulogic_vector:="0010";

    begin
        case EMIF_EPC_CS is
            when EMIF_EPC_IDLE =>
                if ((CE0i and (not LBGNTo)) ='1') then -- no need to check that LBREQ is asserted,
EMIF has priority.
                    EMIF_EPC_NS <= EMIF_EPC_OPEN_BUF;
                else
                    EMIF_EPC_NS <= EMIF_EPC_IDLE;
                end if;

            when EMIF_EPC_OPEN_BUF =>
                if (AREi='1') then       --read cycel
                    EPC_RWo <= '1';
                    EMIF_EPC_NS<=EMIF_EPC_STARTCY;
                elsif (AWEi='1') then     --write cycle
                    EPC_RWo <= '0';
                    EMIF_EPC_NS<=EMIF_EPC_STARTCY;
                else                      -- strobe has not been asserted yet!(SETUP is too long)
```



```
                    EMIF_EPC_NS<=EMIF_EPC_OPEN_BUF;
                end if;

            when EMIF_EPC_STARTCY =>
                EMIF_EPC_NS<=EMIF_EPC_WAIT4RDY;

            when EMIF_EPC_WAIT4RDY =>
                if (RDYi ='1') then
                    EMIF_EPC_NS <= EMIF_EPC_STROBE;
                else
                    EMIF_EPC_NS <= EMIF_EPC_WAIT4RDY; -- wait for EPC to assert RDY
                end if;

            when EMIF_EPC_STROBE =>
                if ((AWEi or AREi) /='1') then  --The write (or read for read access) strobe is
deasserted.
                    EMIF_EPC_NS <= EMIF_EPC_WAIT4B2B;
                else
                    EMIF_EPC_NS <= EMIF_EPC_STROBE; -- Remain in the strobe state, the strobe has
not finished yet, more strobes are required( this would happen if strobe>1 for writes, strobe >3 for
reads)
                end if;

            when EMIF_EPC_WAIT4B2B =>
                if (CE0i /= '1') then  --  end of access
                    EMIF_EPC_NS <= EMIF_EPC_IDLE;
                elsif ((AREi or AWEi)='1') then -- another strobe is strobed, do another access to
EPC
                    EMIF_EPC_NS <= EMIF_EPC_OPEN_BUF;
                else
                    EMIF_EPC_NS <= EMIF_EPC_WAIT4B2B; -- HOLD is too long or EMIF is waiting to
exit a read cycle ( seven minus HOLD before CE is deasserted)
                end if;

            when others =>
                EMIF_EPC_NS <= EMIF_EPC_IDLE;
        end case;
    end process EMIF_EPC_STM;

--------------------------------
--         EPC to HPI         --
--------------------------------
    HDSo     <= EPC_HPI_CS(1) and MASK;
--  HDSo     <= ((EPC_HPI_CS(1) and EPC_RWi)         --read strobe
--               or (EPC_HPI_CS(0) and EPC_HPI_CS(1)) or (EPC_HPI_CS(1) and not EPC_HPI_CS(0) and
CLK));  -- Second strobe in a read de assert after half a cycle


    HCSo     <= (EPC_HPI_CS(2)              -- All the state apart from the IDLE state
                 or EPC_HPI_CS(1)
                 or EPC_HPI_CS(0));

    HCNTLoe  <= HCSo;


    HCNTLo <= ((A19i & (A18i and A19i)) or ('0' & (A2i and not A19i)));

--hcntl_decoding: process(A19i, A18i,A2i)  -- Decoding the HPIA,HPIC,HPID.
--  begin
--      if (A19i='1') then
--          if (A18i='1') then
--              HCNTLo <= "11";  --HPID with increment
--          else
```

```
--               HCNTLo <= "10";  --HPID without increment
--           end if;
--       else
--           if (A2i='1') then
--               HCNTLo <= "01";     --HPIA
--           else
--               HCNTLo <= "00";     --HPIC
--           end if;
--       end if;
--   end process hcntl_decoding;


--   HPI_RWo  <= EPC_RWi; --directly connected
--   HPI_RWoe <= HCSo;

--
--       constant EPC_HPI_IDLE            :std_ulogic_vector:="000";
--       constant EPC_HPI_STARTCY         :std_ulogic_vector:="001";
--       constant EPC_HPI_STROBE1         :std_ulogic_vector:="011";
--       constant EPC_HPI_STROBE2         :std_ulogic_vector:="010";
--       constant EPC_HPI_DEASTROB        :std_ulogic_vector:="100";


EPC_HPI_STM: process(EPC_HPI_CS,LBGNTo,LREQi,BURSTi,EPC_RWi,HRDYi,NOT_LAST,A30i)

    begin
            case EPC_HPI_CS is
                when EPC_HPI_IDLE =>
                    if ((LREQi and LBGNTo and not (A30i)) ='1') then --Access to HPI
                        EPC_HPI_NS <= EPC_HPI_STARTCY;
                    else
                        EPC_HPI_NS <= EPC_HPI_IDLE;
                    end if;
                when EPC_HPI_STARTCY =>
                    EPC_HPI_NS <= EPC_HPI_STROBE1;
                when EPC_HPI_STROBE1 =>
                    if (HRDYi ='1') then
                        EPC_HPI_NS <= EPC_HPI_STROBE2;
                    else
                        EPC_HPI_NS <= EPC_HPI_STROBE1;
                    end if;
                when EPC_HPI_STROBE2 =>
                    EPC_HPI_NS <= EPC_HPI_DEASTROB; -- De assert strobe
                    if (BURSTi='1') then
                        NOT_LAST <='1';             -- there is another 16 bit transfer
                    else
                        NOT_LAST <='0';             -- last 16 bit transfer
                    end if;
                when EPC_HPI_DEASTROB =>
                    if (NOT_LAST ='1') then    -- another 16 bit transfer (a second in a one word
or the first in a next word transfer)
                        EPC_HPI_NS <= EPC_HPI_STROBE1;
                    else
                        EPC_HPI_NS <= EPC_HPI_IDLE;
                    end if;
                when others =>
                    EPC_HPI_NS <= EPC_HPI_IDLE;
            end case;
    end process EPC_HPI_STM;

    MASK_PAR <= ((EPC_HPI_CS(1) and not EPC_HPI_CS(0)) or EPC_SRAM_CS(2)); -- MASK a half a cycle
of the strobe for writes when accessing SRAM or HPI
HALF: process (CLK,EPC_HPI_CS(0),EPC_HPI_CS(1))
```



```
    begin
            if CLK'event and CLK='0' then
                if ((MASK_PAR and (not EPC_RWi)) = '1') then  -- When strobe and the cycle is a
write
                    MASK <= '0';
                else
                MASK <= '1';
                end if;
            end if;
    end process HALF;
end behavior;
```

7.0 Conclusion

The V360EPC PCI Bridge Controller is highly recommended as both PCI master and target bridge for high performance DSP applications. For EMIF (External Memory Interface) to PCI, the V360EPC acts as a local bus target and at the same time as PCI master. For PCI to HPI (Host Processor Interface) or PCI to a shared memory residing on the EMIF bus (SRAM in this application), the EPC acts as PCI target and as the local bus master. Using isolation buffers allows concurrent accesses between PCI to HPI and EMIF to it's external memory devices.

With a small (32 microcell) PLD, the V360EPC easily connects the TMS320C6x DSP EMIF and HPI interfaces to the PCI bus.